

AD-A083 279

CONNECTICUT UNIV STORRS

F/G 9/2

DATA STRUCTURE DEFINITION AND ACCESS CONTROL FACILITIES FOR LAN--ETC(U)

APR 80 B 6 CLAYBROOK

DAA629-78-8-0118

UNCLASSIFIED

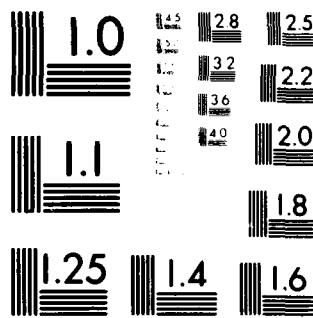
ARO-16264.2-EL

NL

| OF |
PAGE
10



			END
			DATE
			5-80
			DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 19 16264.2-EL	2. SOURCE ACCESSION NO. 18 ARO	3. RECIPIENT'S CATALOG NUMBER 9
4. TITLE (and Subtitle) 6 DATA STRUCTURE DEFINITION AND ACCESS CONTROL FACILITIES FOR LANGUAGES DESIGNED FOR THE DEVELOPMENT OF RELIABLE SOFTWARE		5. FUNDING REPORT & PERIOD COVERED Final Report 1 Sep 78 - 31 Aug 79
7. AUTHOR 10 Billy G. Claybrook		6. PERFORMING ORG. REPORT NUMBER
8. PERFORMING ORGANIZATION NAME AND ADDRESS University of South Carolina Columbia, South Carolina 29208 <i>See TP</i>		9. CONTRACT OR GRANT NUMBER(s) 15 DAAG29-78-G-0118
10. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office P. O. Box 12211 Research Triangle Park, NC 27709		11. REPORT DATE 11 Apr 80
12. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		13. NUMBER OF PAGES 13
LEVEL		14. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) computers data structure computer programs programming languages		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The purpose of the research described here was to develop a data structure definition facility (DSDF) and an access control facility suitable for inclusion in high-level programming languages. The research was not intended to include the design of a complete language but instead involved the development of programming language features that aid in the development of languages designed for producing reliable software. The DSDF was to be capable of specifying and implementing a wide variety of views of data. The intentions were to develop a facility capable of defining		

DTIC
ELECTE
APR 15 1980
SD

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

80 4 15 005

ADA 083279

20. ABSTRACT CONTINUED

real world data objects as well as system-oriented data objects. In addition, the DSDF was to merge the language view of real world and system data objects.

Accession For	
NTIS Grant	<input checked="checked" type="checkbox"/>
DOC T&E	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Joint Publication	<input type="checkbox"/>
By	
Date Received	
Approved for Release	
Dist	Attn/and/or special
A	

Final Report

U. S. Army Research Office Grant No. DAAG29-78-G-0118

entitled

**Data Structure Definition and Access Control
Facilities for Languages Designed for the
Development of Reliable Software**

University of Connecticut, Storrs

September 1, 1978 - August 15, 1979

Billy G. Claybrook

Principal Investigator

80 4 15 005

Problem Studied

The proposed research was to develop a data structure definition facility (DSDF) and an access control facility suitable for inclusion in high-level programming languages. The research was not intended to include the design of a complete language but instead involved the development of programming language features that aid in the development of languages designed for producing reliable software.

The DSDF was to be capable of specifying and implementing a wide variety of views of data. The intentions were to develop a facility capable of defining real world data objects as well as system-oriented data objects. In addition, the DSDF was to merge the language view of real world and system data objects.

Results of the Research

To meet the objectives of this research, an encapsulation mechanism, the module, was designed for specifying and implementing abstract data types (or data objects) in high-level programming languages. The format of the module is illustrated in Figure 1; it is similar in some respects to other encapsulation mechanisms such as the cluster in CLU [3]. Parameterized types are permitted within the module context and restrictions to them, if any, can be specified. The rights specification is an explicit specification of the rights to objects of the type being defined.

The logical structure (LS) component of a module specification essentially characterizes a data object by defining restrictions to relationships. An LS, by itself, does not specify a data type; instead, it highlights features of a data type and communicates them to the user and to the implementer of the type. To some extent, a logical structure specifies what an object looks like, (independent of any representation). In general, a logical structure component of a

```
module module_name [<parameters, if any>]
    restrictions to parameters
    :
    rights
    :
    logical structure
    :
    lsname
    :
    objects
    :
    attributes
    :
    relationships
    :
    invariant assertions
    :
    semantics of operations
    :
    constructive
    :
    nonconstructive
    :
    representation
    :
    implementation
    :
end module_name
```

Figure 1. Format of a module

module can contain multiple logical structure specifications, each one named (using lsname in Figure 1) and giving a different view of the data type. Normally, however, only one logical structure specification is given per data type.

The semantics of operations can be specified in three ways: (1) by using

the constructive approach described in this paper and in Claybrook, et al. [1], (2) by using Gutttag's axioms [2] (only one logical structure specification is meaningful in this case), or (3) by using both the constructive approach and Gutttag's axioms. The representation and implementation specifications are self-explanatory.

The Logical Structure

Basically, the logical structure of a data object is characterized by specifying relationships between constituent object types and by defining restrictions to relationships. Each LS specified in the logical structure component of a module is given a name such as GTREE in the genealogy tree data type specified in Figure 2. A logical structure specification can then be referred to by this name. The name(s) of the component object types are given in the objects section. The attributes of each component object are given in the attributes section; they are given in the form of a function from objects to values. The relationship names and the object type(s) involved in a relationship are given in the relationships section. The relationships are binary relationships and are specified in functional form.

```
module genealogy_tree
  rights  add_person, add_child
  logical structure
    lname GTREE
    objects PERSONS
    attributes  NAME: PERSONS to string
                  DOB: PERSONS to integer
    relationships
      CHILDOF:    PERSONS to PERSONS
      PARENTOF:   PERSONS to PERSONS REDUNDANT
      NEXTALPHA:  PERSONS to PERSONS REDUNDANT
      NEXTOLDEST: PERSONS to PERSONS REDUNDANT
```

Figure 2. A genealogy tree data type.

invariant assertions

- $\forall x, y: \text{PERSONS}$
1. if $x \text{ CHILDOF } y$ then $\text{DOB}(x) > \text{DOB}(y)$
 2. if $x \neq y$ then $\text{NAME}(x) \neq \text{NAME}(y)$
 3. CHILDOF has indegree at most 2
 4. CHILDOF is acyclic
 5. $x \text{ CHILDOF } y$ iff $y \text{ PARENTOF } x$
 6. NEXTALPHA is ordered on NAME and is linear
 7. NEXTOLDEST is ordered on DOB and is linear

semantics of operations

OCCUR = occurrence $\langle P: \text{collection PERSONS}, N: \text{collection NAME},$
 $D: \text{collection DOB}, CO: \text{CHILDOF}, PO: \text{PARENTOF}, NO: \text{NEXTOLDEST},$
 $NA: \text{NEXTALPHA} \rangle$

operations wrt GTREE

add_person (OCCUR, NEWNAME, SOMEDOB) = if $\text{NEWNAME} \in \{N(n) \mid n \in P\}$
then ERROR else $\langle P', N', D', CO, PO, NO', NA' \rangle$ *
where $x: \text{PERSONS}$ and $x \notin P$
 $P' = P \cup \{x\}$
 $N' = N \cup \{ \langle x, \text{NEWNAME} \rangle \}$
 $D' = D \cup \{ \langle x, \text{SOMEDOB} \rangle \}$

end add_person

add_child (OCCUR, NEWNAME, SOMEDOB, PARENT_NAME1, PARENT_NAME2) =
if $\text{NEWNAME} \in \{N(n) \mid n \in P\}$ or $\text{PARENT_NAME1} \notin \{N(n) \mid n \in P\}$
or $\text{PARENT_NAME2} \notin \{N(n) \mid n \in P\}$
or $D(\text{PARENT_NAME1}) > \text{SOMEDOB}$
or $D(\text{PARENT_NAME2}) > \text{SOMEDOB}$ then ERROR
else $\langle P', N', D', CO', PO', NO', NA' \rangle$
where newnode: PERSONS and newnode $\notin P$
 $x \in P \ni \text{PARENT_NAME1} = N(x)$
 $y \in P \ni \text{PARENT_NAME2} = N(y)$
 $P' = P \cup \{\text{newnode}\}$
 $N' = N \cup \{ \langle \text{newnode}, \text{NEWNAME} \rangle \}$
 $D' = D \cup \{ \langle \text{newnode}, \text{SOMEDOB} \rangle \}$
 $CO' = CO \cup \{ \langle \text{newnode}, x \rangle, \langle \text{newnode}, y \rangle \}$
 $PO' = PO \cup \{ \langle x, \text{newnode} \rangle, \langle y, \text{newnode} \rangle \}$

end add_child

end genealogy-tree

Figure 2. (Continued)

*In both the add_person and add_child operations in Figure 2, we indicate that the redundant relations NO and NA are actually affected, even though the operation definitions do not explicitly show this.

The most important ingredient of a logical structure is the invariant assertion. The primary function of the invariant assertion is to specify restrictions to each of the relationships named in the relationships section. In

addition, invariant assertions can also specify relationships between relations (see the genealogy tree data type example in Figure 2 for the relationship between the CHILDOF and PARENTOF relations). The relationships section and the invariant assertions section permit the specifier of a data type to communicate to both the user and the implementer of the data type what he considers to be the type's most important aspects.

The invariant assertions have at least two important uses. First, and perhaps most importantly, they specify what Taylor [4] refers to as the "meaning" of a relationship. For instance, the is-part-of and is-spouse-of relationships are syntactically equivalent but have much different occurrence structures. Secondly, the invariant assertions can be used directly or indirectly to provide a definitive test for valid versus invalid occurrence structures when operations are applied.

The syntax for the assertions is given in Appendix A along with a catalog of properties for relations. Many of these properties are defined using first-order predicate calculus notation. Using names to describe properties makes the invariant assertions easier to read, write and specify.

Semantics of Operations

Previously, we said that the semantics of operations can be specified in three distinct ways: (1) by using the constructive approach described in this paper, (2) by using Guttag's axioms, or (3) by using both the constructive approach and Guttag's axioms.

The operations (constructive approach) are defined in terms of how they affect an occurrence of the data object being specified. An occurrence of a data type is represented as a tuple of elements such as OCCUR in Figure 2. In general, the elements in a tuple consist of collections of instances of all object types, collections of all attribute values, and all relations. For

example, in Figure 2, P is a collection of PERSONS, N is a collection of NAME's, D is a collection of DOB's, and CO, PO, NO, and NA are the four relations restricted by the invariant assertions of the LS named GTREE. An operation definition, then, consists of specifying how the operation affects each of the elements in the occurrence tuple. Not all operations make changes to an occurrence, nor do all operations affect all elements in an occurrence tuple. For example, a find-children operation (not specified) for the genealogy tree data type does not affect an occurrence, and the add_person operation shown in Figure 2 does not change the CHILDOF or PARENTOF relations.

Figure 3 illustrates the stack data type, specified using the principal investigator's constructive approach and Guttag's nonconstructive approach. Redundant specification appears to be useful because Guttag's axioms are useful for verifying that an implementation is correct and the constructive specification is useful as an aid to both the user and the implementer of the type. The utility of redundant specification is a topic of future research.

module stack

rights top, pop, push

logical structure

lname STK

objects NODE

attributes VALUE: NODE to string

relationships ONTOPOF: NODE to NODE

invariant assertions

1. ONTOPOP is linear

semantics of operations

OCCUR = occurrence <N: collection NODE, V: collection VALUE,
O: ONTOPOF>

Figure 3. stack data type specified (partially specified) using both Claybrook's constructive approach and Guttag's nonconstructive approach

operations wrt STK

constructive

```

emptystack() = <∅, ∅, ∅ >
push (OCCUR, NEWVALUE) = <N', V', O'>
  where for x:  NODE and x ∉ N and a ∈ N ⇒ (∃ y ∈ N)(<y, a> ∈ O)
                N' = N ∪ {x}
                V' = V ∪ {<x, NEWVALUE>}
                O' = if OCCUR = emptystack() then ∅
                     else O ∪ {<x, a>}

```

end push

```

pop (OCCUR) = if OCCUR = emptystack() then ERROR
              else <N', V', O'>
  where for x:  NODE, x ∈ N and (∃ a ∈ N)(<a, x> ∈ O)
                N' = N - {x}
                V' = V - {<x, V(x)>}
                O' = O - {<x, a> | <x, a> ∈ O}

```

end pop

```

top (OCCUR) = if OCCUR = emptystack() then ERROR
              else V(x), where x ∈ N and (∃ y ∈ N)(<y, x> ∈ O)

```

end top

nonconstructive

```

declare stk: stack; elm: integer
pop(NEWSTACK) = NEWSTACK
pop(push(stk, elm)) = stk
top(NEWSTACK) = ERROR
top(push(stk, elm)) = elm

```

end stack

Figure 3. (Continued)

Redundant Relations

A relation is classified as redundant if it can be totally specified in terms of the attributes and non-redundant relations. Intuitively, a redundant relation does not provide any new information; it merely highlights a particular aspect of the logical structure. NEXTALPHA, NEXTOLDEST and PARENTOF relations are redundant in the genealogy tree logical structure (see Figure 2). CHILDOF is not redundant since it provides new information which cannot be obtained from

the attributes. Note that since CHILDOF and PARENTOF are almost interchangeable, one could have chosen CHILDOF as the redundant relation and PARENTOF as the non-redundant relation. Non-redundant relations must be included in each operation definition, whereas redundant relations need not be included in the operation definition. In some cases, the specifier may choose to define how an operation actually affects a relation, even though the relation is redundant. This approach assures the implementer of all changes that an operation makes to all relations, including redundant relations.

Summary of Results

The significant accomplishments of the year's research efforts include:

- 1) the specification of the module encapsulation mechanism as a means for specifying and implementing abstract data types,
- 2) the development of the logical structure component of the module, in particular the invariant assertions, for specifying restrictions to relationships between constituent object types, and
- 3) the development of the notation for specifying the semantics of operations (using the constructive approach to specification).

The combination of these three things provide the basis for a number of further research topics, including verifying the correctness of implementations of abstract data types. *

* This is one of the objectives of the renewal year of this grant.

References

1. Claybrook, Billy G., et al. "Logical Structure Specification and Data Type Definition," Proceedings of ACM 79 Conference, October 1979, pp. 203-211.
2. Guttag, John V. et al. "Abstract Data Types and Software Validation," CACM, Vol. 21, December 1978, pp. 1043-1064
3. Liskov, Barbara, et al. "Abstraction Mechanisms in CLU," CACM, Vol. 20, August 1977, pp. 564-576.
4. Taylor, Robert W. "Observations on the Attributes of Database Sets," Data Base Description, Douque, B. C. and Nijssen, G. M. (eds.), North-Holland, Amsterdam 1975, pp. 73-84.

Appendix A

This appendix specifies the syntax for invariant assertions and presents a catalog of names that are used to expedite and facilitate the specification of invariant assertions.

An assertion is of the form:

<Relation name> <noise words> <property>

ON <object set>

or assertion in first-order predicate calculus augmented by standard set notations involving the named objects, attributes, and relationships.

Notes

<Relation name> is any relation named in the relationships section of the LS.

<noise words> may be added for readability.

<property> may have embedded parameters and are defined in the following catalog.

ON <object> is optional. <object> is the name of a set of objects named in the objects section of the LS.

The default value is the set of objects on which relation is defined.

In the following catalog, R stands for the relation parameter and A stands for the object set parameter. Embedded parameters are underlined.

noloops $(\forall a:A)(\underline{\text{not } aRa})$.

nocycles $(\forall a:A)(\forall n:\text{integer})(\exists b_1, b_2, \dots, b_n:A)$
 $(aRb_1 \text{ and } b_1Rb_2 \text{ and } \dots \text{ and } b_nRa \text{ and } a \neq b \text{ and } n \geq 1)$.

acyclic R has noloops and R has nocycles.

indegree n $(\forall a:A)(\text{indegree}(a)=n).$ *

outdegree n $(\forall a:A)(\text{outdegree}(a)=n).$ *

indegree at most n $(\forall a:A)(\text{indegree}(a) \leq n).$ *

outdegree at most n $(\forall a:A)(\text{outdegree}(a) \leq n).$ *

$n:m$ correspondence R has indegree at most n and R has outdegree at most m .

reflexive $(\forall a:A)(aRa)$

symmetric $(\forall a:A)(\forall b:A)(aRb \text{ iff } bRa).$

anti-symmetric $(\forall a,b:A)(aRb \text{ and } bRa \text{ implies } a=b).$

transitive $(\forall a,b,c:A)(\text{if } aRb \text{ and } bRc \text{ then } aRc).$

partition R is reflexive, symmetric and transitive.

partially ordered R is reflexive, anti-symmetric and transitive.

totally ordered R is partially ordered and $(\forall a,b:A)(aRb \text{ or } bRa).$

linear R is acyclic with $(\text{card}\{A\} \leq 1 \text{ or } (\exists a \in A)(\text{indegree}(a) = 0 \text{ and } \text{outdegree}(a) = 1 \text{ and } (\exists b \in A)(\text{indegree}(b) = 1 \text{ and } \text{outdegree}(b) = 0 \text{ and } (\forall c:A)(\text{if } c \neq a \text{ and } c \neq b \text{ then } \text{indegree}(c) = \text{outdegree}(c) = 1.))$

ordered on x R is linear and $(\forall a,b:A)(aRb \text{ implies } x(a) < x(b)).$

tree $A \neq \emptyset \text{ or } (\exists a:A)(\text{indegree}(a) = 0 \text{ and } (\forall b:A)(\text{if } a \neq b \text{ then } \text{indegree}(b) = 1))$
and R is acyclic.

pair $(\exists a,b:A)(A = \{a,b\} \text{ and } R = \{<a,b>\}).$

star $(\exists a:A)(\text{indegree}(a) = 0 \text{ and } (\forall b:A)(\text{if } a \neq b \text{ then } \text{indegree}(b) = 1 \text{ and } \text{outdegree}(b) = 0)).$

set of Y $(\forall P)(\text{if } P \subseteq A \text{ and } (\forall x:P)(\exists a:A)(a \notin P \text{ and } (aRx \text{ or } xRa)) \text{ then } R|P \text{ is } Y \text{ on } P)).$

forest R is a set of tree.

pairs R is a set of pair.

stars R is a set of star.

* $\text{indegree}(a) = \text{card}\{b | aRb\}$ and $\text{outdegree}(a) = \text{card}\{b | bRa\}.$

Publications

"Logical Structure Specification and Data Type Definition," Proceedings
of the ACM 79 Conference, October 1979, pp. 203-211.

Personnel

Billy G. Claybrook, Principal Investigator (12 months)

Donald Criscione, Graduate Research Assistant (11 months)

Craig Cleaveland, Consultant (1 month)